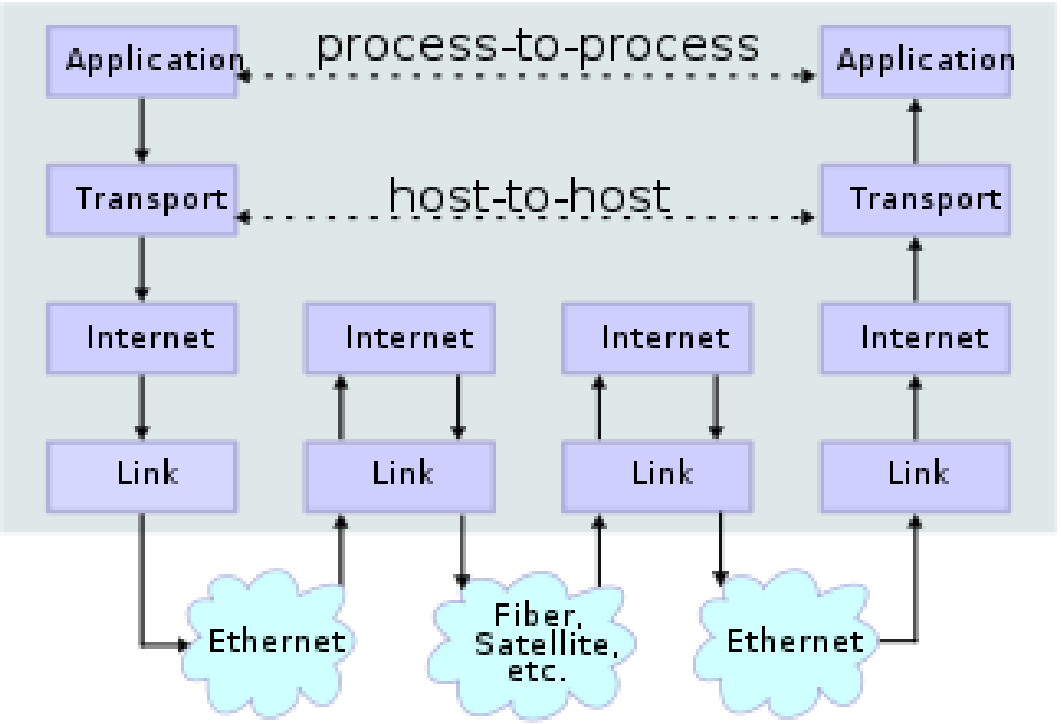
Advanced Programming

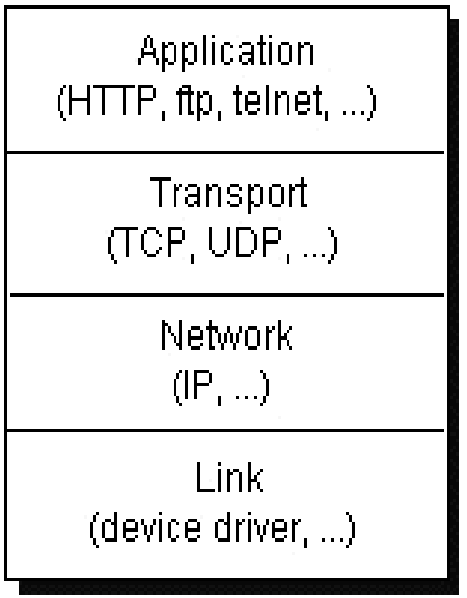
Networking

Protocol

Protocol - A **set of rules** governing the exchange or transmission of data between devices.



When you write Java programs that communicate over the network, you are programming at the application layer



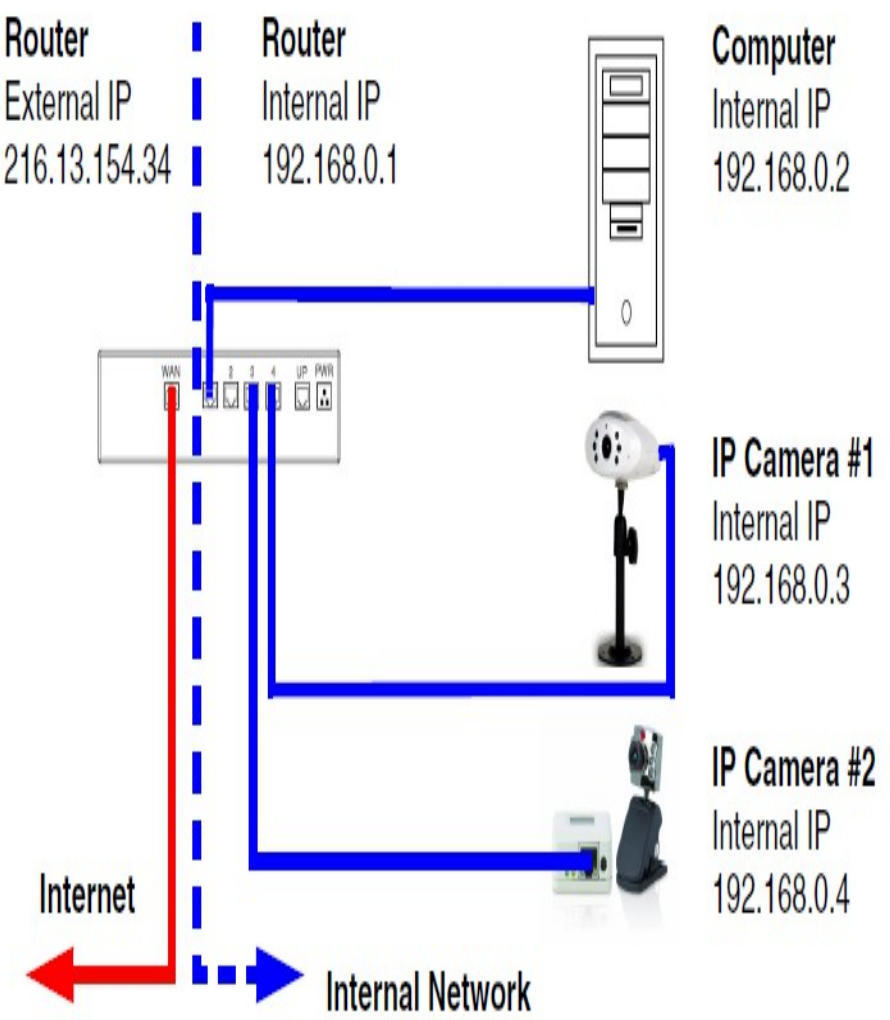
Internet Address

java.net.InnetAddress

- *Innet4Address* (32-bit)
  - ✔ 85.122.23.145 - numerical
  - ✔ fenrir.info.uaic.ro - symbolic
- *Innet6Address* (128-bit)
  2002:0:0:0:0:0:557a:1791

Address Types

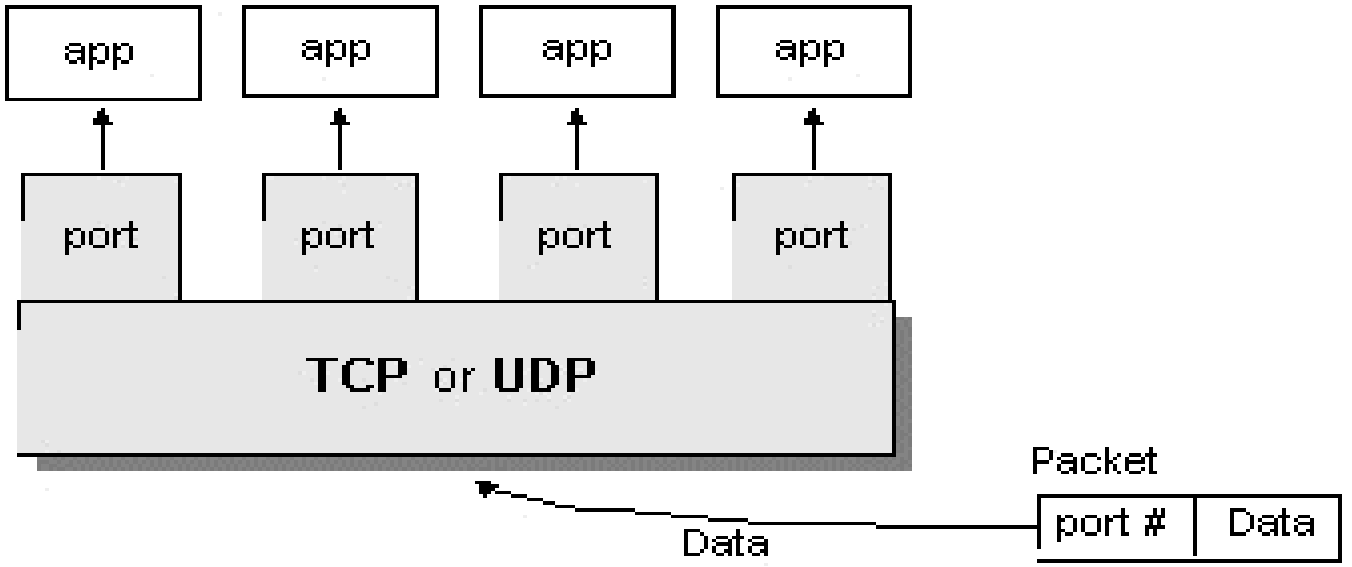
- unicast
- multicast
- localhost (127.0.0.1)



Port

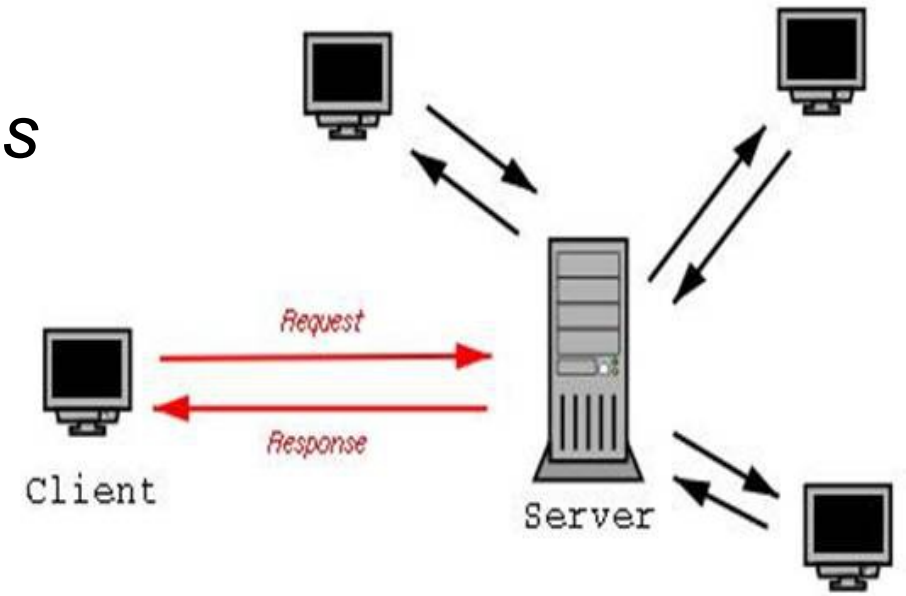
A **port** is a **16-bit number**, which uniquely identifies a process offering services over the network.

- Possible values: 0 − 65535
- Reserved values: 0 − 1023 (the *well-known* ports)



The Client-Server Model

- **The Server**
  - ofers some network *services*
  - runs at a specified *port*
  - must be able to handle many clients *concurrently*



- **The Client**
  - initiate the conversation with the server
  - must know the *IP address* and the *port* of the server
  - sends *requests* and receive *responses*

Sockets

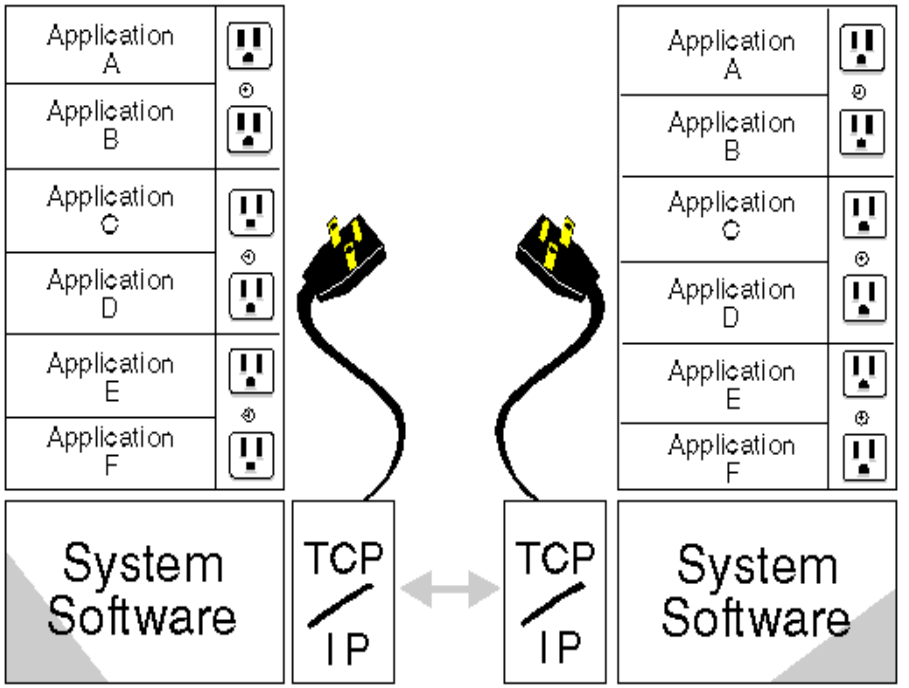
Socket - A software abstraction describing one end-point of a two-way communication link between two programs running on the network.

- TCP: *Socket, ServerSocket*
- UDP: *DatagramSocket*

java.net.InetSocketAddress

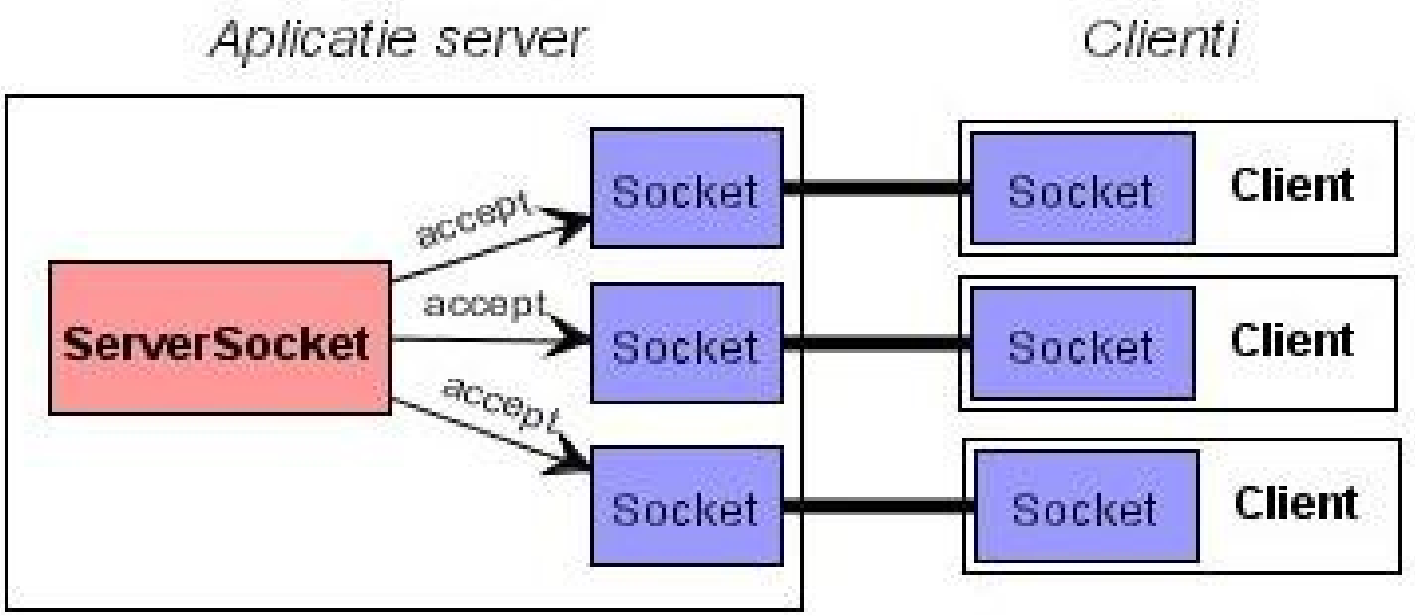
(IP adress, port)

(hostName, port)



Communication Over TCP

- *Transport Control Protocol*
- **Connection-based**
- **Reliable** flow of data between two computers



A Simple TCP Server

```
public class SimpleServer {
  // Define the port on which the server is listening
  public static final int PORT = 8100;

  public SimpleServer() throws IOException {
    ServerSocket serverSocket = null ;
    try {
      serverSocket = new ServerSocket(PORT);
      while (true) {
        System.out.println ("Waiting for a client ...");
        Socket socket = serverSocket.accept();
        // Execute the client's request in a new thread
        new ClientThread(socket).start();
      }
    } catch (IOException e) {
      System.err. println ("Ooops... " + e);
    } finally {
      serverSocket.close();
    }
  }
  public static void main ( String [] args ) throws IOException {
    SimpleServer server = new SimpleServer ();
  }
}
```

Creating the Response

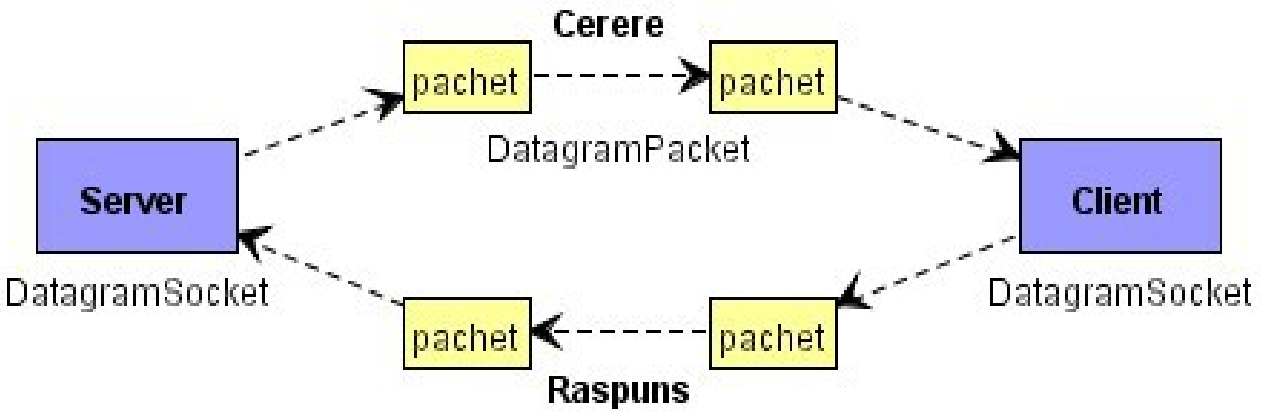
```
class ClientThread extends Thread {
  private Socket socket = null ;
  public ClientThread (Socket socket) { this.socket = socket ; }
  public void run () {
    try {
      // Get the request from the input stream: client → server
      BufferedReader in = new BufferedReader(
          new InputStreamReader(socket.getInputStream()));
      String request = in.readLine();

      // Send the response to the oputput stream: server → client
      PrintWriter out = new PrintWriter(socket.getOutputStream());
      String raspuns = "Hello " + request + "!";
      out.println(raspuns);
      out.flush();
    } catch (IOException e) {
      System.err.println("Communication error... " + e);
    } finally {
      try {
        socket.close(); // or use try-with-resources
      } catch (IOException e) { System.err.println (e); }
    }
  }
}
```

A Simple TCP Client

```
public class SimpleClient {
  public static void main (String[] args) throws IOException {
    String serverAddress = "127.0.0.1"; // The server's IP address
    int PORT = 8100; // The server's port
    try (
      Socket socket = new Socket(serverAddress, PORT);
      PrintWriter out =
                new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader (
             new InputStreamReader(socket.getInputStream())) ) {

      // Send a request to the server
      String request = "World";
      out.println(request);

      // Wait the response from the server ("Hello World!")
      String response = in.readLine ();
      System.out.println(response);

    } catch (UnknownHostException e) {
      System.err.println("No server listening... " + e);
    }
  }
}
```

Communication Over UDP

- *User Datagram Protocol*
- **Independent** packets of data, called **datagrams**
- **NOT** connection-based
- **No guarantees** about arrival or order of delivery



A Simple UDP Server

```
int portServer = 8200; // Server's port

// Create a server side communication socker
DatagramSocket socket = new DatagramSocket(portServer);

// Wait for incoming package
byte buf[] = new byte[256];
DatagramPacket request = new DatagramPacket(buf, buf.length );
socket.receive(request);

// Get the address and the port of the client who sent the request
InetAddress clientAddress = request.getAddress();
int clientPort = request.getPort();

// Create the response
String message = "Hello " + new String(request.getData()) + "!";
buf = message.getBytes();

// Send a response package to the client
DatagramPacket response =
    new DatagramPacket(buf, buf.length, clientAddress, clientPort);
socket.send(response);
```

A Simple UDP Client

```
InetAddress serverAddress = InetAddress.getByName("127.0.0.1");
int serverPort = 8200;

// Create a client-side communication socket
// The socket is bound to any available port on the local host machine
DatagramSocket socket = new DatagramSocket();

// Create and send a request package
byte buffer1[] = "World".getBytes();
DatagramPacket request =
  new DatagramPacket(buffer1, buffer1.length, serverAddress, serverPort);
socket.send(request);

// Wait for the response
byte buffer2[] = new byte[256];
DatagramPacket response = new DatagramPacket(buffer2, buffer2.length );
socket.receive(response);

// Here it is: Hello World!
System.out.println(new String(response.getData()));
```

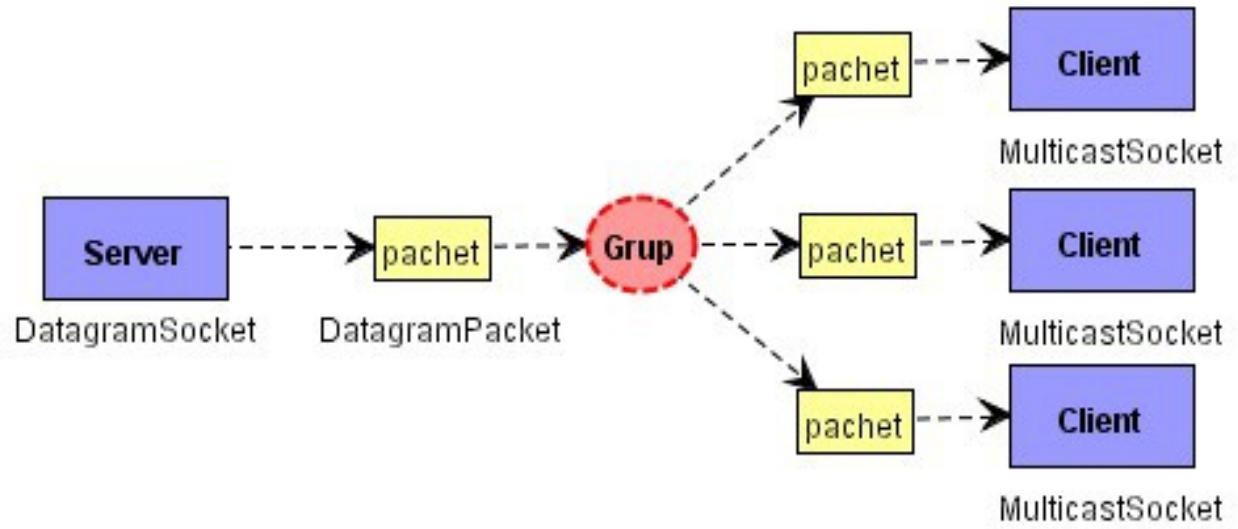
PortUnreachableException may be thrown if the socket is connected to a currently unreachable destination.
Note, there is no guarantee that the exception will be thrown.

Sending Datagrams to a Group

A **multicast** group of clients is specified by a class D IP address:
224.0.0.1- 239.255.255.255

When one sends a message to a multicast group, all subscribing recipients to that host and port receive the message.

```
InetAddress group = InetAddress.getByName("230.0.0.1");
// Join the party...
MulticastSocket clientSocket = new MulticastSocket();
clientSocket.joinGroup(group);
```



Communication Over HTTP

The Hypertext Transfer Protocol → Communication for the World Wide Web

URL = Uniform Resource Locator

- Static resources (HTML pages, texts, images, etc.)

```
http://profs.info.uaic.ro/~acf/java/slides/en/networking_slide_en.pdf
```

- Dynamic resources (servlets, JSP/PHP pages, etc.)

```
http://85.122.23.145:8080/WebApplication/hello.jsp
```

A URL can be broken into several parts:

- The protocol: *http*
- The host machine: *profs.info.uaic.ro, 85.122.23.145*
- The port of the inner TCP connection: *default (80), 8080*
- The *path* to the component is both protocol dependent and host dependent

Working With URLs

java.net.URL

Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine.

```
try {
  URL url = new URL("https://docs.oracle.com/javase/8/docs/api/");
} catch (MalformedURLException e) {
  System.err.println("Invalid URL: " + e);
}
```

- **Query** the URL object
- **Read the contents** of the URL
- **Conect** to the URL

Reading the Contents of an URL

```
public class URLContentReading {
  public static void main(String[] args) throws IOException {

    String resource = "http://profs.info.uaic.ro/~acf/hello.txt";
    BufferedReader reader = null ;
    try {
      URL url = new URL (resource);
      InputStream in = url.openStream();
      reader = new BufferedReader(new InputStreamReader(in));

      // Read the contents of the URL, line by line
      String line;
      while (( line = reader. readLine ()) != null ) {
        System.out.println (line);
      }
    } catch ( MalformedURLException e) {
      System.err.println ("Invalid URL: " + e);
    } finally {
      if (reader != null) reader. close ();
    }
  }
}
```

```
// Using streams
String text = reader.lines().collect(Collectors.joining("\n"));
```

Parsing a JSON Response

JavaScript Object Notation. Format for storing and exchanging data.
An easier-to-use alternative to XML.

```
String resource = "http://api.icndb.com/jokes/random";
InputStream in = new URL(resource).openStream();
BufferedReader reader =
  new BufferedReader(new InputStreamReader(in));

String json = reader.lines().collect(Collectors.joining("\n"));
/* { "type": "success",
     "value": {
         "id": 546,
         "joke": "Chuck Norris does infinit loops in 4 seconds.",
         "categories": ["nerdy"]
     }
   }*/
// We use Google Gson library
Gson gson = new Gson();
Map<String, Object> map = new HashMap<>();
map = (Map<String, Object>) gson.fromJson(json, map.getClass());

 Map<String, Object> value =
    (Map<String, Object>) map.get("value");

System.out.println(value.get("joke"));
```

Connecting to a URL

Establishing a a 2-way communications link between the application and a URL

```
public class URLConnectionDemo {
  public static void main(String[] args) throws IOException {

    URL url = new URL("http://localhost:8080/App/HelloWorld);
    URLConnection connection = url.openConnection();
    connection.setDoOutput(true);

    OutputStreamWriter out =
        new OutputStreamWriter(connection.getOutputStream());
    String param = URLEncoder.encode("Duke & World", "UTF-8");
    out.write("string=" + param);
    out.close();

    BufferedReader in = new BufferedReader(
        new InputStreamReader(connection.getInputStream()));
    String response;
    while ((response = in.readLine()) != null) {
      System.out.println(response);
    }
    in.close();
  }
}
```

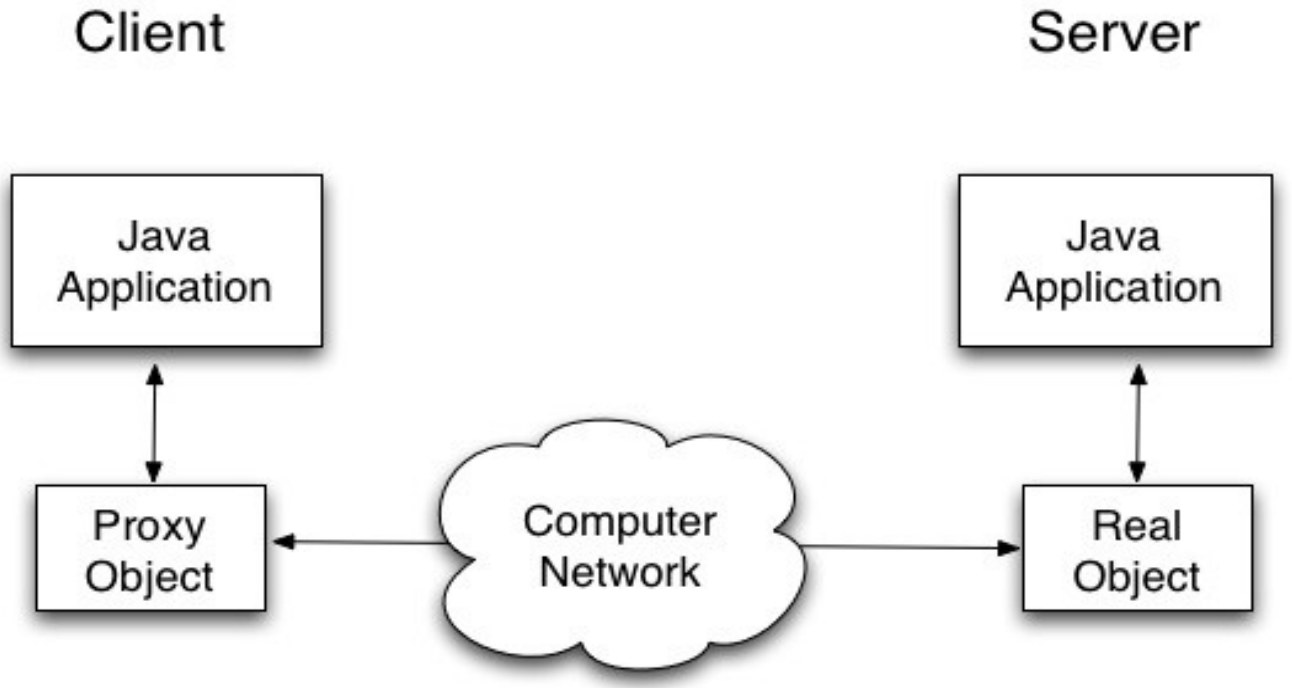
Remote Method Invocation (RMI)

- Higher level network programming
- Allows objects running in one Java Virtual Machine to invoke methods objects running in another JVM
- **Distributed object applications**

  A *server* program creates remote objects, makes references to these objects accessible, and waits for clients to invoke their methods.
- Syntax and semantics **similar** to standard applications
- Issues:
  - ✔ How to locate and identify remote objects?
  - ✔ How to send arguments and receive results?
  - ✔ How to handle remote exceptions?
  - ✔ What about garbage collection?

Remote Proxy

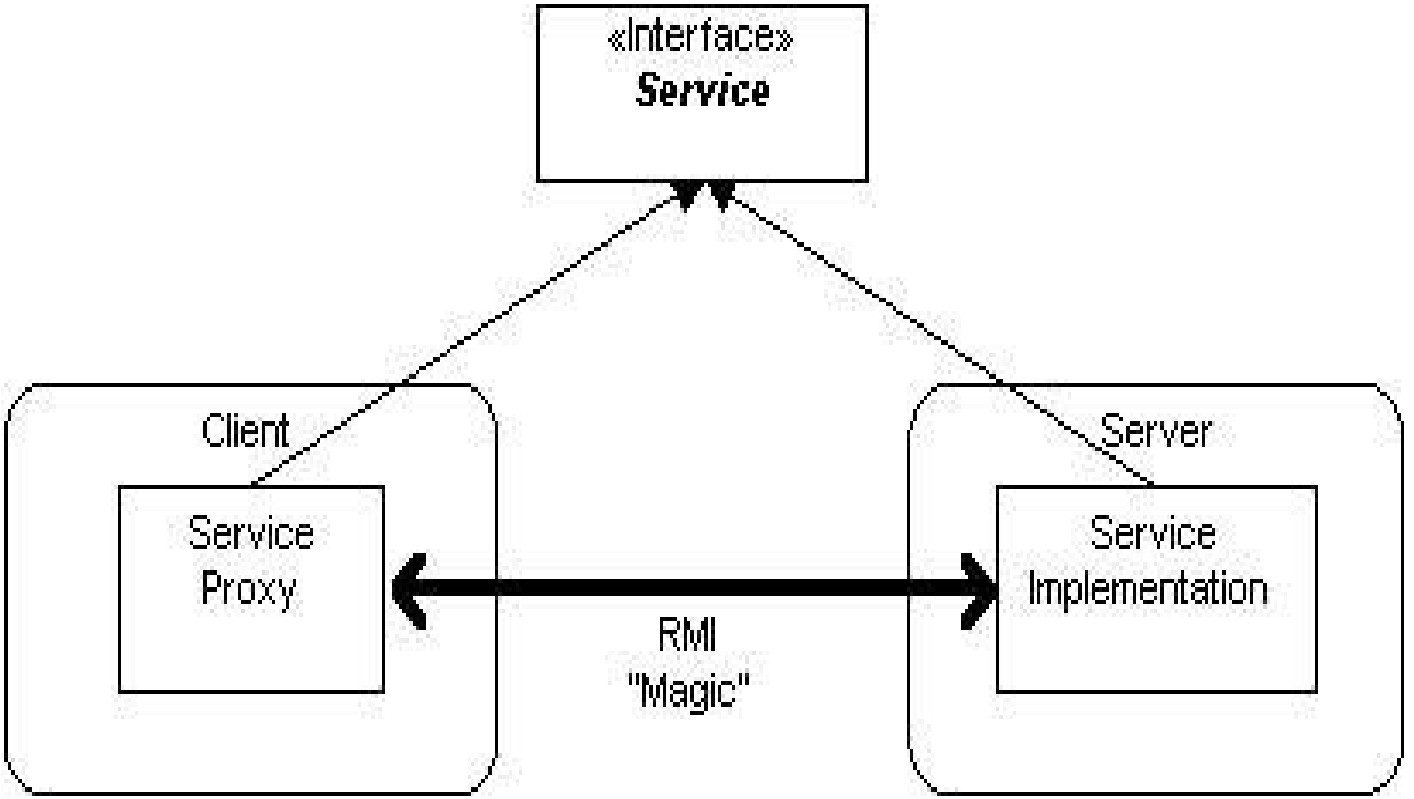
- **Proxy** - An object which acts as an interface to another object (also called *surrogate*, *placeholder*)
- **Remote Proxy** - A local object representing a *remote* object (one that belongs to a different address space).



What other types of Proxy do you know?

RMI Basic Principle

The separation between **behavior** and **implementation**



Identifying Remote Objects

Name Services

- **JNDI** (Java Naming and Directory Interface)
- **RMI Registry** (JAVA-HOME/

The common name services operations:

- ***bind*** - the association between an object and a symbolic name
- ***lookup*** - obtaining the reference to an object using its symbolic name

RMI "*Hello World!*"

Hello.java → **Interface** describing the service

Must be available to both server and client.

```
package service;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
  String sayHello(String name) throws RemoteException;
}
```

The Service Implementation

HelloImpl.java → Server-side implementation of the interface

```
package server;

import java.rmi.RemoteException;
import service.Hello;
public class HelloImpl implements Hello {
  public HelloImpl() throws RemoteException {
    super();
  }
  public String sayHello(String name) {
    return "Hello " + name + " !";
  }
}
```

Exposing the Service

HelloServer.java

```
package server;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;
import service.Hello;
public class HelloServer {
  public static void main(String[] args) throws Exception {
    Hello hello = new HelloImpl();
    Hello stub = (Hello) UnicastRemoteObject.exportObject(hello, 0);
    Registry registry = LocateRegistry.getRegistry();
    registry.bind("Hello", stub);
    System.out.println("Hello Service activated!");
  }
}
```

The Client

HelloClient.java

```
package client;

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import service.Hello;

public class HelloClient{

  public static void main(String[] args) throws Exception {

    Registry registry = LocateRegistry.getRegistry("localhost");

    Hello hello = (Hello) registry.lookup("Hello");

    String response = hello.sayHello("World");

    System.out.println(response);

  }

}
```